Adapting Component-based Systems at Runtime via Policies with Temporal Patterns

Olga Kouchnarenko, Jean-Francois Weber

Adapting Component-based Systems at Runtime via Policies with Temporal Patterns*

Olga Kouchnarenko[1,2] and Jean-François Weber[1]

[1] FEMTO-ST CNRS and University of Franche-Comté, Besançon, France
{okouchnarenko,jfweber}@femto-st.fr
[2] Inria/Nancy-Grand Est, France

Abstract. Dynamic reconfiguration allows adding or removing components of component-based systems without incurring any system downtime. To satisfy specific requirements, adaptation policies provide the means to dynamically reconfigure the systems in relation to (events in) their environment. This paper extends event-based adaptation policies by integrating temporal requirements into them. The challenge is to reconfigure component-based systems at runtime while considering both their functional and non-functional requirements. We illustrate our theoretical contributions with an example of an autonomous vehicle location system. An implementation using the Fractal component model constitutes a practical contribution. It enables dynamic reconfigurations guided by either enforcement or reflection adaptation policies.

1 Introduction

Dynamic reconfiguration is a mechanism that allows components of component-based systems to be added to or removed without incurring any system downtime. The challenge is to build or maintain trustworthy systems which satisfy both functional and non-functional requirements.

Let us illustrate the adaptation and reconfiguration needs on a characteristic example inspired from a real case study in the land transportation domain. The example concerns the Cybercar concept, a public transport system with automated driving capabilities. Within the autonomous vehicle case study, a location composite component — a critical part of land transportation systems — is made up of different positioning systems, like GPS or Wi-Fi. Thanks to adaptation policies, the location composite component architecture can be modified to use either GPS, Wi-Fi, or GPS+Wi-Fi positioning systems, depending on some non-functional properties, such as available energy.

Recent implementations to support the development of component-based systems, like those for the Fractal reference implementation Julia[3] or for GCM[4], tend to provide mechanisms for the execution of high-level adaptation policies.

* This work has been partially funded by the Labex ACTION, ANR-11-LABX-01-01.

[3] http://fractal.objectweb.org/julia/index.html

[4] http://gridcomp.ercim.org/

Adaptation polices implemented in Tangram4Fractal [8], triggered by qualitative expressions of fuzzy logic (e.g., "`power is` *low*"), do not allow expressing temporal constraints. In [9,11], the authors have introduced a component-based system model equipped with either *a*) adaptation policies using qMEDL[5] logic [16] or *b*) a linear temporal logic, called FTPL[6], expressing architectural constraints, events, and temporal patterns [11]. FTPL, based on Dwyer's work on patterns and scopes [13], and being more expressive than qMDEL at providing temporal schemas, this paper proposes to bridge the gap between [9] and [11].

Our main contribution is the use of FTPL logic for triggering adaptation policies and specifying behaviours of the system under scrutiny. As a practical contribution, we have implemented these more expressive adaptation policies to guide and control dynamic reconfigurations via enforcement and reflection adaptation policies. When a violation of a property is detected, the reflection's purpose is to reconfigure the system to mitigate, if possible, the failure, whereas the enforcement aims to circumvent property violations.

Furthermore, as temporal properties often cannot be evaluated to true or false during the system execution, and so cannot, *a fortiori*, the extended policies, this paper addresses this question by evaluating at runtime, in a progressive manner, both temporal properties and extended policies. To this end, like in RV-LTL [4], in addition to *true* and *false* values, *potential true* and *potential false* values are used whenever an observed behaviour has not yet led to an acceptance or a violation of the property under consideration.

Layout of the paper. Section 2 introduces our motivating example, a component model, and its operational semantics, while Section 3 covers a temporal pattern logic over reconfiguration sequences. In Sect. 4, linear temporal patterns are integrated into adaptation policies. The evaluation at runtime of both temporal properties and extended—reflection and enforcement—adaptation policies is presented in Sect. 5. We show that these mechanisms guarantee a system's behaviour allowed by the initial system specification (correctness result). Finally, an implementation allowing the user to deal with the Fractal component model is described in Sect. 6. Section 7 presents our conclusion.

2 Motivating Example and Background

Component models can be very heteregeneous. Most of them consider software components that can be seen as black boxes (or grey boxes if some of their inner features are visible) having fully described interfaces. Behaviours and interactions are specified using components' definitions and their interfaces. In this section, after introducing a motivating example, we revisit the architectural reconfiguration model introduced in [11,21]. In general, the system configuration is the specific definition of the elements that define or prescribe what a system is

[5] qMEDL is a flavor of MEDL used to express quantity of resource properties.

[6] FTPL stands for TPL (Temporal Pattern Language) prefixed by 'F' to denote its relation to Fractal-like components and to first-order integrity constraints over them.

composed of, while a reconfiguration can be seen a transition from a configuration to another.

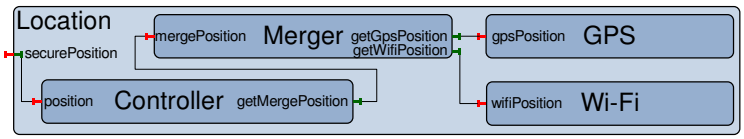

Fig. 1. The location component in Fractal

Motivating Example. The development, validation, and certification of a new type of urban vehicles with fully or partially automated driving capabilities, like CyCab [2] or Cristal[7] aimed at replacing the private car, are a challenging issue. These distributed and embedded systems require the expression of functional as well as non-functional properties, for example time-constrained response, QoS, and availability of required services.

A positioning system is a critical part of a land transportation system. Many positioning systems have been proposed over the past few years. Among them, we can mention GPS, GALILEO or GLONASS positioning systems which belong to the Global Navigation Satellite Systems (GNSS, for short). Other localisation systems have been designed using various technologies, like Wireless personal networks such as Bluetooth, sensors, GNSS repeaters, or visual landmarks.

Figure 1 gives an abstract view of a composite location component developed within the Fractal component framework. This component includes several positioning systems, like GPS or Wi-Fi, a controller, and a merger. Each positioning system is composed of an atomic positioning component and a software component to validate perceived data. The validation components transfer the positioning data to the merger if they are precise enough. The merger applies a particular algorithm to merge data obtained from positioning systems. Finally, the controller's purpose is to request and to acknowledge the receipt of positioning data.

Moreover, there is a need to make the system's architecture evolve at runtime. Reconfigurations, however, must not happen at any but suitable circumstances. The location composite component architecture can be modified to use, e.g., either GPS, or Wi-Fi, or GPS+Wi-Fi positioning systems, depending on some non-functional properties, such as available energy, and the events from the current indoor/outdoor environment. For example, the following requirement "After the GPS component has been removed, the level of energy has to be greater than 33% before this component is added back" makes use of temporal and architectural constraints to allow the "with GPS" reconfiguration. Then, thanks to adaptation policies, several possible reconfigurations can be determined,

[7] `http://www.projet-cristal.net/`

and the most suitable reconfiguration can be chosen. For example, when the available energy makes both reconfigurations "with GPS" and "with GPS+Wi-Fi" possible within an adaptation policy, this policy can be used to put system's priorities to the "with GPS+Wi-Fi" reconfiguration, for more reliability.

Configurations. Following [11], we define a configuration to be a set of architectural elements (components, required or provided interfaces, and parameters) together with relations to structure and to link them.

Definition 1 (Configuration). *A configuration c is a tuple $\langle Elem, Rel \rangle$ where*

- $Elem = Components \uplus Interfaces \uplus Parameters \uplus Types$ *is a set of architectural elements, such that*
  - $Components$ *is a non-empty set of the core entities, i.e components;*
  - $Interfaces = RequiredInts \uplus ProvidedInts$ *is a finite set of the (required and provided) interfaces;*
  - $Parameters$ *is a finite set of component parameters;*
  - $Types = ITypes \uplus PTypes$ *is a finite set of the interface types and the parameter data types;*
- $Rel = \begin{cases} Container \uplus ContainerType \uplus Contingency \\ \uplus\ Parent \uplus Binding \uplus Delegate \uplus State \uplus Value \end{cases}$
  is a set of architectural relations which link architectural elements, such that
  - $Container : Interfaces \uplus Parameters \to Components$ *is a* total function *giving the component which supplies the considered interface or the component of a considered parameter;*
  - $ContainerType : Interfaces \uplus Parameters \to Types$ *is a* total function *that associates a type to each (required or provided) interface and to each parameter;*
  - $Contingency : RequiredInts \to \{mandatory, optional\}$ *is a* total function *indicating whether each required interface is mandatory or optional;*
  - $Parent \subseteq Components \times Components$ *is a* relation *linking a sub-component to the corresponding composite component*[8]*;*
  - $Binding : ProvidedInts \to RequiredInts$ *is a* partial function *which binds together a provided interface and a required one;*
  - $Delegate : Interfaces \to Interfaces$ *is a* partial function *to express delegation links;*
  - $State : Components \to \{started, stopped\}$ *is a* total function *giving the status of instantiated components;*
  - $Value : Parameters \to \{t | t \in PType\}$ *is a* total function *which gives the current value of each parameter.*

Example 1. To illustrate our model, the example of Fig. 1 is described in Fig. 2.

[8] For any $(p, q) \in Parent$, we say that q has a sub-component p, i.e. p is a child of q. Shared components (sub-components of multiple enclosing composite components) can have more than one parent.

$$
\begin{array}{ll}
Components & = \{controller, gps, location, merger, wifi\} \\
ProvidedInts & = \{gpsPosition, mergePosition, position, securePosition, wifiPosition\} \\
RequiredInts & = \{getGpsPosition, getMergePosition, getWifiPosition\} \\
Parameters & = \{GpsPowerUsage, Power, Trust, WifiPowerUsage\} \\
Types & = \{MergePosition, Position, SecurePosition, int\} \\
Container & = \{gpsPosition \mapsto gps, mergePosition \mapsto merger, position \mapsto controller, \\
 & \quad securePosition \mapsto location, wifiPosition \mapsto wifi, \\
 & \quad getGpsPosition \mapsto merger, getMergePosition \mapsto controller, \\
 & \quad getWifiPosition \mapsto merger, GpsPowerUsage \mapsto merger, \\
 & \quad Power \mapsto controller, Trust \mapsto merger, WifiPowerUsage \mapsto merger\} \\
ContainerType & = \{getGpsPosition \mapsto Position, getMergePosition \mapsto MergePosition, \\
 & \quad getWifiPosition \mapsto Position, gpsPosition \mapsto Position, \\
 & \quad mergePosition \mapsto MergePosition, position \mapsto SecurePosition, \\
 & \quad securePosition \mapsto SecurePosition, wifiPosition \mapsto Position, Trust \mapsto int, \\
 & \quad GpsPowerUsage \mapsto int, Power \mapsto int, WifiPowerUsage \mapsto int\} \\
Contingency & = \{getGpsPosition \mapsto optional, getMergePosition \mapsto mandatory, \\
 & \quad getWifiPosition \mapsto optional\} \\
Parent & = \{(controller, location), (gps, location), (merger, location), (wifi, location)\} \\
Binding & = \{gpsPosition \mapsto getGpsPosition, mergePosition \mapsto getMergePosition, \\
 & \quad wifiPosition \mapsto getWifiPosition\} \\
Delegate & = \{position \mapsto securePosition\} \\
State & = \{controller \mapsto started, gps \mapsto started, location \mapsto started, merger \mapsto started, \\
 & \quad wifi \mapsto started\} \\
Value & = \{GpsPowerUsage \mapsto 3, Power \mapsto 95, Trust \mapsto 0, WifiPowerUsage \mapsto 2\}
\end{array}
$$

Fig. 2. Configuration of the example of Fig. 1

We also introduce a set CP of configuration propositions on the architectural elements and the relations between them. These properties are specified using first-order logic formulae [17]. The interpretation of functions, relations, and predicates over *Elem* is done according to basic definitions in [17] and Def. 1.

Let $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ be a set of configurations. We introduce an *interpretation* function $l : \mathcal{C} \to CP$ which gives the largest conjunction of $cp \in CP$ evaluated to true on $c \in \mathcal{C}$. We say that a configuration $c = \langle Elem, Rel \rangle$ satisfies $cp \in CP$, written $[\![\ c \models cp\ ]\!] = \top$, when $l(c) \Rightarrow cp$. In this case, cp is valid on c. Otherwise, c does not satisfy cp, written $[\![\ c \models cp\ ]\!] = \bot$.

Among all the configuration propositions, there are constraints common to all the component-based system architectures. They define *consistent* configurations. For example, two bound interfaces must have the same interface type and their suppliers must be sub-components of the same composite. These consistency constraints are respectively expressed by $\forall\ ip \in ProvidedInts, ir \in RequiredInts.$ $(Binding(ip) = ir \Rightarrow ContainerType(ip) = ContainerType(ir))$, and
$\forall\ ip \in ProvidedInts, ir \in RequiredInts.$
$\Big(Binding(ip) = ir \Rightarrow \Big(\exists\ c \in Components. \begin{pmatrix} (Container(ip), c) \in Parent \\ \wedge (Container(ir), c) \in Parent \end{pmatrix}\Big)\Big)$
The reader interested in consistency constraints is referred to [21].

Reconfigurations. Reconfigurations make the component-based architecture evolve dynamically. They are combinations of primitive operations such as instantiation/destruction of components; addition/removal of components; binding/unbinding of component interfaces; starting/stopping components; setting parameter values of components. The normal running of different components also changes the architecture, e.g., by modifying parameter values or stopping

components. Let $\mathcal{R}_{run} = \mathcal{R} \cup \{run\}$ be a set of evolution operations, where $\mathcal{R}$ is a finite set of reconfiguration operations, and run is the name of a generic action used to represent all the running operations of the component-based system.

Definition 2 (Reconfiguration model). *The operational semantics of component-based systems with reconfigurations is defined by the labelled transition system* $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ *where* $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ *is a set of configurations,* $\mathcal{C}^0 \subseteq \mathcal{C}$ *is a set of initial configurations,* $\rightarrow \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ *is the reconfiguration relation, and* $l : \mathcal{C} \rightarrow CP$ *is a total interpretation function.*

Let us note $c \stackrel{ope}{\rightarrow} c'$ for $(c, ope, c') \in \rightarrow$, and $c \stackrel{ope}{\rightarrow}$ when there is a target configuration c' such that $c \stackrel{ope}{\rightarrow} c'$. Given the model $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$, an evolution path (or a path for short) σ of S is a sequence of configurations $c_0, c_1, c_2, \ldots$ such that $\forall i \geq 0.\ \exists\ ope_i \in \mathcal{R}_{run}.(c_i \stackrel{ope_i}{\rightarrow} c_{i+1})$. We write $\sigma(i)$ to denote the i-th configuration of σ. The notation σ_i denotes the suffix path $\sigma(i), \sigma(i+1), \ldots$, and σ_i^j denotes the segment path $\sigma(i), \sigma(i+1), \ldots, \sigma(j-1), \sigma(j)$. Let Σ denote the set of paths, and Σ^f $(\subseteq \Sigma)$ the set of finite paths. A configuration c' is reachable from c when there is a path $\sigma = c_0, c_1, \ldots, c_n$ in Σ^f s.t. $c = c_0$ and $c' = c_n$. An execution is a path σ in Σ s.t. $\sigma(0) \in \mathcal{C}^0$.

3 FTPL: a Temporal Logic for Dynamic Reconfigurations

In this section, we briefly recall the FTPL logic introduced in [11]. Inspired by [3,1], we present a new *progressive* semantics for FTPL properties evaluation at runtime, where, unlike [10], the evaluation of a trace or temporal property at any given state of a path σ is based on its evaluation at the previous state.

3.1 Syntax and Notations

Basically, constraints on the architectural elements and the relations between them are specified as configuration propositions defined in Sect. 2. In addition, the proposed logic contains external events, as well as events from reconfiguration operations, temporal properties, and, finally, trace properties embedded into temporal properties. Let $Prop_{FTPL}$ denote the set of the FTPL formulae obeying the FTPL grammar given below. Let us first give the FTPL syntax.

<FTPL> ::= <tpp> | <events> | *cp*
<tpp> ::= **after** <events><tpp> | **before** <events><trp> | <trp> **until** <events> | <trp>
<trp> ::= **always** *cp* | **eventually** *cp* | <trp> ∧ <trp> | <trp> ∨ <trp>
<events> ::= <event>,<events> | <event>
<event> ::= *ope* **normal** | *ope* **exceptional** | *ope* **terminates** | *ext*

In order to give the semantics for these formulae, we introduce the set $\mathbb{B}_4 = \{\bot, \bot^p, \top^p, \top\}$, where $\bot, \top$ stand resp. for *false* and *true* values, and $\bot^p, \top^p$ for *potential false* and *potential true* values. As in [4], we consider $\mathbb{B}_4$ together with the truth non-strict ordering relation $\sqsubseteq$ satisfying $\bot \sqsubseteq \bot^p \sqsubseteq \top^p \sqsubseteq \top$. On

$\mathbb{B}_4$ we define the unary operation $\neg$ as $\neg\bot = \top$, $\neg\bot^p = \top^p$, $\neg\top^p = \bot^p$, $\neg\top = \bot$, and we define two binary operations $\sqcup$, $\sqcap$ resp. as the minimum and maximum interpreted wrt. $\sqsubseteq$. Thus, $(\mathbb{B}_4, \sqsubseteq)$ is a finite *de Morgan* lattice but not a Boolean lattice.

3.2 FTPL Basic Semantics

FTPL semantics is basic for events and configuration propositions, and runtime-oriented for other properties. We write $[\![\sigma(i) \models cp]\!]$ to denote the evaluation of the configuration proposition cp in $\mathbb{B}_4$[9] at the i-th configuration of the path σ.

External events (like events in [20]) occur instantaneously and can be seen as invocations of methods performed by (external) sensors when a change is detected in their environment. For each external event ext that may occur on a given execution path σ, we define *a*) a guard cp_{ext}, which is a first-order logic formula over the parameters specified in the invocation of the method ext, and *b*) an assertion $eval_\sigma$, valued in $\mathbb{B}_2$. Intuitively, if, at or before the i-th and after the $i-1$-th state (or, if $i = 0$, at the first state) of an execution path σ, there is at least one occurrence of ext s.t. $cp_{ext} = \top$ then $eval_\sigma(cp_{ext}, i) = \top$, otherwise $eval_\sigma(cp_{ext}, i) = \bot$.

The following definition present FTPL semantics for *a*) reconfiguration events —"*ope* **normal**" (resp. "*ope* **exceptional**") when a reconfiguration *ope* terminates normally (resp. abnormally) or "*ope* **terminates**" when *ope* terminates regardless of its result—, *b*) external events, and *c*) lists of events. We write $[\![\sigma(i) \models e]\!]$ to denote the evaluation of the event (resp. list of events) e in $\mathbb{B}_4$ at the i-th configuration of the path σ.

Definition 3 (FTPL Events Semantics). *Let ope be a reconfiguration operation, ext an external event, e an event, and events a list of events.*

The interpretation of the events at the i-th state of the path σ is defined by:

$$[\![\sigma(i) \models ope\ \textbf{normal}]\!] = \begin{cases} \top & \text{if } i > 0 \wedge \sigma(i-1) \neq \sigma(i) \wedge \sigma(i-1) \overset{ope}{\rightarrow} \sigma(i) \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\sigma(i) \models ope\ \textbf{exceptional}]\!] = \begin{cases} \top & \text{if } i > 0 \wedge \sigma(i-1) = \sigma(i) \wedge \sigma(i-1) \overset{ope}{\rightarrow} \sigma(i) \\ \bot & \text{otherwise.} \end{cases}$$

$$[\![\sigma(i) \models ope\ \textbf{terminates}]\!] = [\![\sigma(i) \models ope\ \textbf{normal}]\!] \sqcup [\![\sigma(i) \models ope\ \textbf{exceptional}]\!]$$

$$[\![\sigma(i) \models ext]\!] = eval_\sigma(cp_{ext}, i)$$

$$[\![\sigma(i) \models e, events]\!] = [\![\sigma(i) \models e]\!] \sqcup [\![\sigma(i) \models events]\!]$$

3.3 FTPL *Progressive* Semantics

Let $\sigma \in \Sigma$ be a path. Given an FTPL property from $Prop_{FTPL}$, its value on σ is given by the interpretation function $[\![\_ \models \_]\!] : \Sigma \times Prop_{FTPL} \to \mathbb{B}_4$ defined below by induction. In order to evaluate, in a *progressive* fashion, FTPL expressions at runtime, without consulting a complete history of FTPL properties'

[9] Since $\mathbb{B}_2 \subset \mathbb{B}_4$, the evaluation $[\![c \models cp]\!]$ of the configuration proposition $cp \in CP$ on the configuration c detailed on p. 5 is considered to be valued in $\mathbb{B}_4$.

evaluation (like in [10]), we introduce the following notations. Let $\phi_\sigma = [\![\sigma \models \phi]\!]$ be the evaluation of an FTPL formula where ϕ is a list of events, a trace property, or a temporal property. We denote $\phi_\sigma(i)$ the evaluation of ϕ on σ, at the i-th state of the path.

Furthermore, following [13], if the scope of an FTPL property ϕ is restricted to the suffix path σ_k, $k \geq 0$, we write $\phi_{\sigma_k} = [\![\sigma_k \models \phi]\!]$ for such a restriction, and $\phi_{\sigma_k}(i)$ for the evaluation in $\mathbb{B}_4$ of this restriction at the i-th state of σ, where $i \geq k$. Then, the evaluation of ϕ on the path σ ($\phi_\sigma = [\![\sigma \models \phi]\!]$), is similar to the evaluation of ϕ on the suffix path σ_0 starting at the first configuration, wich is $\phi_{\sigma_0} = [\![\sigma_0 \models \phi]\!]$. For the sake of simplicity, we also write $cp_{\sigma_k}(i) = [\![\sigma_k(i) \models cp]\!]$.

Definition 4 (FTPL Runtime Progressive Trace Properties Semantics). *Let cp be a configuration proposition, ϕ (resp. φ) a trace property of the form $\phi =$ **always** cp (resp. $\varphi =$ **eventually** cp). We define $\phi_{\sigma_k}(i)$ (resp. $\varphi_{\sigma_k}(i)$), the evaluation in $\mathbb{B}_4$ of $[\![\sigma_k \models \phi]\!]$ (resp. $[\![\sigma_k \models \varphi]\!]$) at the i-th state of σ when the scope is restricted to σ_k, by:*

- *for $i = k$, $\phi_{\sigma_k}(k) = \top^p \sqcap cp_\sigma(k)$; $\varphi_{\sigma_k}(k) = \bot^p \sqcup cp_\sigma(k)$*
- *for $i > k$, $\phi_{\sigma_k}(i) = \phi_{\sigma_k}(i-1) \sqcap cp_\sigma(i)$; $\varphi_{\sigma_k}(i) = \varphi_{\sigma_k}(i-1) \sqcup cp_\sigma(i)$*

Furthermore, let ψ_1 and ψ_2 be two trace properties, then:
$[\![\sigma_k \models \psi_1 \wedge \psi_2]\!] = [\![\sigma_k \models \psi_1]\!] \sqcap [\![\sigma_k \models \psi_2]\!]$; $[\![\sigma_k \models \psi_1 \vee \psi_2]\!] = [\![\sigma_k \models \psi_1]\!] \sqcup [\![\sigma_k \models \psi_2]\!]$

On the scope starting at the k-th state of σ, if at state k one has $cp_\sigma(k) = \top$ (resp. $cp_\sigma(k) = \bot$), the trace property **always** cp (resp. **eventually** cp) is evaluated to $\top^p$ (resp. $\bot^p$); otherwise, it is evaluated to $\bot$ (resp. $\top$). Then, for $i > k$, at the i-th state of σ, **always** cp (resp. **eventually** cp) is evaluated to the minimum (resp. maximum), interpreted wrt. $\sqsubseteq$, of *a*) its evaluation at the previous state and *b*) $cp_\sigma(i)$. Table 1 shows an example of the evaluation of such trace properties.

Table 1. Evaluation of trace properties

Let be $\phi =$ **always** $\neg cp$ and $\varphi =$ **eventually** cp

i	k	$k+1$	$k+2$	$k+3$	$k+4$	$k+5$	$k+6$	$k+7$	$k+8$	$\cdots$
$cp_\sigma(i)$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\cdots$
$\phi_{\sigma_k}(i)$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\top^p$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\varphi_{\sigma_k}(i)$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\bot^p$	$\top$	$\top$	$\top$	$\top$	$\top$

Definition 5 (FTPL Runtime Progressive Lists of Events Semantics). *Let e be a list of events. We define $e_{\sigma_k}(i)$, the evaluation in $\mathbb{B}_4$ of $[\![\sigma_k \models e]\!]$ at the i-th state of σ when the scope is restricted to σ_k, by:*

- *for $i = k$, $e_{\sigma_k}(k) = [\![\sigma_k(k) \models e]\!]$*
- *for $i > k$, $e_{\sigma_k}(i) = [\![\sigma_k(i) \models e]\!] \sqcup (\top^p \sqcap e_{\sigma_k}(i-1))$*

Intuitively, the expression $[\![\sigma(i) \models e]\!] \sqcup (\top^p \sqcap e_{\sigma_k}(i-1))$ evaluates to $\top$ if there is an occurrence of e at configuration i, and to $\bot$ (resp. $\top^p$) if there is no occurrence of e at configuration i and no (resp. at least one) occurrence of e happening before configuration i on the scope starting at configuration k.

Definition 6 (FTPL Runtime Progressive Temporal Properties Semantics). *Let tpp be a temporal property, trp a trace property, e a list of events, ϕ (resp. φ, ψ) a temporal property of the form $\phi =$ **after** e tpp (resp. $\varphi =$ **before** e trp, $\psi = trp$ **until** e). We define $\phi_{\sigma_k}(i)$ (resp. $\varphi_{\sigma_k}(i)$, $\psi_{\sigma_k}(i)$), the evaluation in $\mathbb{B}_4$ of $[\![\sigma_k \models \phi]\!]$ (resp. $[\![\sigma_k \models \varphi]\!]$, $[\![\sigma_k \models \psi]\!]$) at the i-th state of σ when the scope is restricted to σ_k, by: for $i \geq k$,*

$$\phi_{\sigma_k}(i) = \Big(\prod_{j \in \mathcal{I}_{\sigma_k^i}(e)} tpp_{\sigma_j}(i) \Big) \sqcap \top^p \quad \begin{array}{l} where\ \mathcal{I}_{\sigma_k^i}(e) = \{j | k \leq j \leq i \wedge [\![\sigma(j) \models e]\!] = \top\} \\ represents\ the\ set\ of\ indexes\ for\ an\ occurrence\ of\ e. \end{array}$$

$$\varphi_{\sigma_k}(i) = \begin{cases} \top^p & if\ e_{\sigma_k}(i) = \bot \vee i = k \\ \bot & if\ e_{\sigma_k}(i) = \top \wedge trp_{\sigma_k}(i-1) \in \{\bot, \bot^p\} \\ \varphi_{\sigma_k}(i-1) & otherwise \end{cases}$$

$$\psi_{\sigma_k}(i) = \begin{cases} \top^p & if\ trp_{\sigma_k}(i) \neq \bot \wedge e_{\sigma_k}(i) = \top \wedge\ e_{\sigma_k}(i-1) = \bot \wedge\ trp_{\sigma_k}(i-1) \in \{\top^p, \top\} \\ \bot^p & if\ trp_{\sigma_k}(i) \neq \bot \wedge (e_{\sigma_k}(i) = \bot \vee i = k) \\ \bot & if\ trp_{\sigma_k}(i) = \bot \vee (e_{\sigma_k}(i) = \top \wedge\ trp_{\sigma_k}(i-1) \in \{\bot, \bot^p\}) \\ \psi_{\sigma_k}(i-1) & otherwise \end{cases}$$

By definition, the evaluation of $\phi =$ **after** *e tpp* is either *a*) $\top^p$ as long as *e* does not occur or if *tpp* is evaluated to $\top^p$ or $\top$ on each suffix of the path starting at an occurrence of *e*, or *b*) $\bot$ if on any of these suffixes *tpp* is evaluated to $\bot$, or *c*) $\bot^p$, otherwise.

For $\varphi =$ **before** *e trp*, its evaluation is either *a*) $\top^p$ if *e* has not occurred yet, or *b*) $\bot$ if for each occurrence of *e*, *trp* is evaluated to $\bot$ or $\bot^p$ on the segment starting at the beginning of the considered scope and ending at the previous $i-1$-th configuration on the σ path. Otherwise, ϕ at the *i*-th configuration is evaluated to its value at the previous $i-1$-th configuration.

Intuitively, the $\psi = trp$ **until** *e* property can be seen as being evaluated similarly to **before** *e trp*, but with the two following exceptions: *a*) when *trp* is evaluated to $\bot$, ψ is evaluated to $\bot$; otherwise, *b*) on the beginning part of the scope and as long as *e* has not occurred, ψ is evaluated to $\bot^p$.

Finally, we say that a reconfiguration model $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ satisfies a property $\phi \in Prop_{FTPL}$, written $S \models \phi$, *if* $\forall \sigma. (\sigma \in \Sigma(S) \wedge \sigma(0) \in \mathcal{C}^0 \Rightarrow \phi_\sigma = \top)$.

Table 2 shows the evaluation of the temporal property ϕ which is always $\bot^p$ except on and after the configuration when the event *entry* occurs until the configuration preceding the occurrence of the event *exit* where it is $\top^p$. Note that the event *entry* occurs at both the *j*-th and the *l*-th configurations whereas the evaluation of $e = start, exit$ is $\top$ at configurations 0 and *k*, hence $\phi_{\sigma_0}(i) = \varphi_{\sigma_0}(i)$ for $i < k$, and $\phi_{\sigma_0}(i) = \varphi_{\sigma_0}(i) \sqcap \varphi_{\sigma_k}(i)$ for $i \geq k$.

3.4 FTPL Expressiveness

We should note that FTPL trace properties are either *a*) a subset of safety properties, as **always** *cp*, or, *b*) a subset of guarantee properties, as **eventually** *cp*, or *c*) conjunctions and disjunctions of properties from these subsets (safety and guarantee properties). Consequently, according to the *safety-progress* hierarchy [24,7], they are included in obligation properties which represent a subset of response properties. In [15] the issue of enforceable properties, originally

Table 2. Detail of the evaluation of "**after** $start, exit$ ($\top^p$ **until** $entry$)"

Let be $e = start, exit$, $\varphi = \top^p$ **until** $entry$, $\phi =$ **after** $start, exit$ ($\top^p$ **until** $entry$) = **after** e φ

i	0	1	$\cdots$	$j-1$	j	$j+1$	$\cdots$	$k-1$	k	$k+1$	$\cdots$	$l-1$	l	$l+1$	$\cdots$
$[\![\sigma(i) \models start]\!]$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$
$[\![\sigma(i) \models entry]\!]$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\cdots$
$[\![\sigma(i) \models exit]\!]$	$\bot$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$
$[\![\sigma(i) \models e]\!]$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\cdots$	$\bot$	$\bot$	$\bot$	$\cdots$
$\mathcal{I}_{\sigma_0^i}(e)$	$\{0\}$	$\{0\}$	$\cdots$	$\{0\}$	$\{0\}$	$\{0\}$	$\cdots$	$\{0\}$	$\{0,k\}$	$\{0,k\}$	$\cdots$	$\{0,k\}$	$\{0,k\}$	$\{0,k\}$	$\cdots$
$entry_{\sigma_0}(i)$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\top^p$	$\cdots$	$\top^p$	$\top^p$	$\top^p$	$\cdots$	$\top^p$	$\top$	$\top^p$	$\cdots$
$\varphi_{\sigma_0}(i)$	$\bot^p$	$\bot^p$	$\cdots$	$\bot^p$	$\top^p$	$\top^p$	$\cdots$	$\top^p$	$\top^p$	$\top^p$	$\cdots$	$\top^p$	$\top^p$	$\top^p$	$\cdots$
$entry_{\sigma_k}(i)$	$\times$	$\times$	$\cdots$	$\times$	$\times$	$\times$	$\cdots$	$\times$	$\bot$	$\bot$	$\cdots$	$\bot$	$\top$	$\top^p$	$\cdots$
$\varphi_{\sigma_k}(i)$	$\times$	$\times$	$\cdots$	$\times$	$\times$	$\times$	$\cdots$	$\times$	$\bot^p$	$\bot^p$	$\cdots$	$\bot^p$	$\top^p$	$\top^p$	$\cdots$
$\phi_{\sigma_0}(i)$	$\bot^p$	$\bot^p$	$\cdots$	$\bot^p$	$\top^p$	$\top^p$	$\cdots$	$\top^p$	$\bot^p$	$\bot^p$	$\cdots$	$\bot^p$	$\top^p$	$\top^p$	$\cdots$

addressed because of infinite sequences, is extended to finite and infinite properties at runtime. It is then established that enforceable properties are exactly response properties. Hence, FTPL trace properties, as a subset of obligation properties, are enforceable as well.

Before ending this section, let us mention (infinite) renewal properties [22], a superset of safety properties also containing some liveness properties, that can be enforced by *edit-automata* as runtime monitors. Intuitively, a property is a *renewal property* if every valid infinite sequence of actions has infinitely many valid prefixes. This is exactly the case of response properties [7]. FTPL trace properties being, as established above, response properties, they are also renewal properties and can then be enforced by edit-automata. Consequently, FTPL temporal properties, acting as scopes [13] of trace properties, can also be enforced in the same way.

4 Integrating Temporal Properties into Adaptation Policies

Although one of the main advantages of reconfigurable component-based systems is the ability of the system's architecture to evolve at runtime, reconfigurations must not happen at any but in suitable circumstances. In order to supervise and to dynamically influence component-based systems reconfigurations, this section introduces adaptation policies indicating reconfigurations suitable to perform, and rules that can impact on the architecture of the component-based system model.

To take into account some resource constraints, events in the system environment, or even properties over sequences of reconfigurations, we propose to extend adaptation policies by integrating FTPL properties into them. For that, adaptation policies exploit the above-mentioned properties and their domains. Each domain defines its specific vocabulary to qualify associated properties, based on the evaluation of the architectural or temporal constraints. Adaptation policies are defined by: *a*) architectural reconfiguration operations to specify the

possible modifications of the architecture; and *b*) adaptation rules to link the properties concerning the component-based system and the need[10] to activate a reconfiguration. We adapt definitions in [8,9] to fit in with our component-based system model semantics, when extending them with temporal properties.

Definition 7 (Adaptation Policies). *Let S be a reconfiguration model, and $Ftype$ a set of fuzzy types. Given $\sigma(i) \in \mathcal{C}$, a finite set AP of adaptation policies for $\sigma(i)$ is composed of elements $A = \langle R_N, R_R \rangle$, where:*

- *$R_N \subseteq \mathcal{R}$ is a finite (non-empty) set of architectural reconfiguration names,*
- *$R_R = \{\langle F, B, G, I \rangle\}$ is a finite (non-empty) set of adaptation rules, where*
  - *$F \in Ftype$ is a fuzzy type,*
  - *$B \subseteq \{\phi_\sigma(i) = value \mid \phi \in Prop_{FTPL} \wedge value \in \mathbb{B}_4\}$ is a set of properties in $Prop_{FTPL}$ evaluated in $\mathbb{B}_4$ on $\sigma(i)$,*
  - *$G \subseteq \{cp_\sigma(i) = value \mid cp \in CP \wedge value \in \mathbb{B}_2\}$ is a set of configuration propositions in CP evaluated in $\mathbb{B}_2$ on $\sigma(i)$,*
  - *$I \subseteq R_N \times F$ is a relation between reconfigurations and fuzzy values.*

Let us denote $B_{\sigma(i)}$ (resp. $G_{\sigma(i)}$) the conjunction of the properties evaluations in B (resp. guards evaluations in G) on $\sigma(i)$.

To illustrate adaptation policies with events, let us suppose that the system where the location component is running can dynamically support the removal or the addition of either the GPS and the Wi-Fi components. Of course, at any given time there should be at least one of these components present. In certain cases, however, it can be beneficial to remove one of these components.

For example, when the energy level of the vehicle is low, the Wi-Fi component can be removed, and then added back when the internal batteries are recharged. Furthermore, when the vehicle enters a "Wi-Fi area" where there is no GPS signal available, it is suitable to remove the GPS component, which can be added back after exiting such an area. Figure 3 displays the `cycabgps` adaptation policy, which is written using a syntax inspired by Tangram4Fractal [8] adaptation policies. This policy influences the `addgps` and `removegps` reconfigurations to respectively add or remove the GPS component. It uses three events (lines 3 to 5): *start* (that occurs only when the adaptation policy becomes effective) and *entry* (resp. *exit*) that occurs when the vehicle enters (resp. exits) a "Wi-Fi area".

Example 2. For the adaptation policy in Fig. 3, we have the architectural reconfigurations set $R_N = \{\texttt{addgps}, \texttt{removegps}\}$ and $Ftype = \{\{\texttt{low}, \texttt{medium}, \texttt{high}\}\}$ which contains all the fuzzy types used in this policy. For the adaptation rule spanning lines 23 to 25, we have, using the notation of Def. 7, $F = \{\texttt{low}, \texttt{medium}, \texttt{high}\}$, $B = \{\textbf{after}\ start, exit\ (\top^p\ \textbf{until}\ entry) = \top^p\}$, $G = \{gps \in Components \wedge wifi \in Components = \top\}$, and finally $I = \{(\texttt{removegps}, \texttt{high})\}$. This adaptation rule expresses that when the expression in B holds (i.e., the vehicle is within a "Wi-Fi area" - cf. Table 2 for details of the evaluation), if both the GPS and the Wi-Fi components are present, then the utility of removing the GPS component, by invoking the `removegps` reconfiguration, is high.

[10] As in [8,9], we use a fuzzy value (e.g.; in {`low`, `medium`, `high`}) to express this need.

```
policy cycabgps

  event entry
  event exit
  event start

 when (after start,exit (
      P_TRUE4 until entry))=P_TRUE4
  if (gps in Components) = FALSE
  then utility of addgps is low

 when (Power < 33) = TRUE4
  if (gps in Components) = FALSE
  then utility of addgps is low

 when (Power < 33) = FALSE4
  if (gps in Components) = FALSE
  then utility of addgps is high

 when (Power < 33) = FALSE4
  if (gps in Components and
       wifi in Components) = TRUE
  then utility of removegps is low

 when (after start,exit (
      P_TRUE4 until entry))=P_TRUE4
  if (gps in Components and
       wifi in Components) = TRUE
  then utility of removegps is high

 when (Power < 33) = TRUE4
  if (gps in Components and
       wifi in Components) = TRUE
  then utility of removegps is high

end policy
```

Fig. 3. `cycabgps` adaptation policy

Let $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$ be a reconfiguration model and AP_S a finite set of adaptation policies for S. We now define how the adaptation policies affect the behaviour of the component-based system model.

Definition 8 (Restriction by Adaptation Policies). *The restriction of S by adaptation policies in AP_S is defined as $S \triangleleft AP_S = \langle \mathcal{C} \triangleleft AP_S, \mathcal{C}^0 \triangleleft AP_S, \mathcal{R}_{run}, \rightarrow, l \rangle$, where $\mathcal{C} \triangleleft AP_S$ is the least set s.t. if $c \in \mathcal{C}$ and $A \in AP_S$ then $c \triangleleft A \in \mathcal{C} \triangleleft AP_S$, $\mathcal{R}_{run} \cap (\cup_{A \in AP_S} R_N) \neq \emptyset$, $l : \mathcal{C} \triangleleft AP_S \rightarrow CP$ is a total interpretation function, and for every $ope \in \mathcal{R}_{run}$, the transition relation $\rightarrow \in \mathcal{C} \triangleleft AP_S \times \mathcal{R}_{run} \times \mathcal{C} \triangleleft AP_S$ is the least set of triples $(c \triangleleft A, ope, c' \triangleleft A)$ satisfying the following rules:*

$$[ACT1] \quad \frac{c \xrightarrow{ope} c'}{c \triangleleft A \xrightarrow{ope} c' \triangleleft A} \quad (ope \in \bigcup_{A \in AP_S} R_N) \wedge B_c \wedge G_c$$

$$[ACT2] \quad \frac{c \xrightarrow{ope} c'}{c \triangleleft A \xrightarrow{ope} c' \triangleleft A} \quad ope \notin \bigcup_{A \in AP_S} R_N$$

This definition means that all the configurations in $\mathcal{C} \triangleleft AP_S$ are reachable from initial configurations by either reconfiguration operations obeying adaptation policies (Rule *[ACT1]*), or by normal reconfigurations which are not involved in the adaptation policy (Rule *[ACT2]*).

5 Runtime Policy Evaluation

Given a component-based system and a set of adaptation policies, a problem occurring while applying adaptation policies is to ensure that the reconfigurations (of a component-based system obeying the policies) conform to the specified reconfigurations. More formally, for two component-based systems modelled by S and $S \triangleleft AP_S$, the problem is to decide whether the behaviour of S obeying its adaptation policies in AP_S is also a behaviour of S. To address this problem, we propose to use the ready simulation notion [6].

Definition 9 (Ready Simulation). *Let S_1 and S_2 be two reconfiguration models over $\mathcal{R}_{run}$. A binary relation $\simeq \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ is a ready simulation iff, for all ope in $\mathcal{R}_{run}$, $(c_1, c_2) \in \simeq$ implies*

i) Whenever $(c_1, ope, c_1') \in \rightarrow_1$, then there exists $c_2' \in \mathcal{C}_2$ such that $(c_2, ope, c_2') \in \rightarrow_2$ and $(c_1', c_2') \in \simeq$.
ii) Whenever $c_1 \overset{ope}{\not\rightarrow}$, then $c_2 \overset{ope}{\not\rightarrow}$.

We say that S_1 and S_2 are ready-similar, written $S_1 \simeq S_2$, if $\forall c_1^0 \in \mathcal{C}_1^0 \exists c_2^0 \in \mathcal{C}_2^0 . (c_1^0, c_2^0) \in \simeq$. Following [6], we keep the ready-set definition for S as $readies(c) = \{ope \mid ope \in \mathcal{R}_{run} \wedge c \overset{ope}{\rightarrow}\}$. A useful fact follows immediately from Def. 9: $(c_1, c_2) \in \simeq$ implies $readies(c_1) = readies(c_2)$. Consequently, it is enough to show the disequality of the ready-sets to show that the ready simulation does not hold between two configurations.

To be able to establish whether $S \triangleleft AP_S \simeq S$ or not, and thus to provide a correctness result concerning the restriction by adaptation policies, we consider the following decision problem.

Adaptation Problem.
Input: Component-based system modelled by $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \rightarrow, l \rangle$, $c \in \mathcal{C}$, and the set $AP \subseteq AP_S$ of adaptation policies for c.
Output: true if $\forall A \in AP, c \triangleleft A \simeq c$, and **false** otherwise.

For the component-based system under its adaptation policies, we define the ready set wrt. Def. 8 by: $readies(c \triangleleft A) = \{ope \mid ope \in \mathcal{R}_{run} \setminus \cup_{A \in AP} R_N \wedge c \overset{ope}{\rightarrow}\} \cup \{ope \mid (ope \in \bigcup_{A \in AP} R_N) \wedge B_c \wedge G_c \wedge c \overset{ope}{\rightarrow}\}$. Then, again, it is easy to see that $c \triangleleft A \simeq c$ implies $readies(c \triangleleft A) = readies(c)$. Both S and $S \triangleleft AP_S$ being infinite state systems, the simulation problem is undecidable in general. However, when the ready sets are different, we can reach a conclusion. Consequently,

Proposition 1. *The adaptation problem is semi-decidable.*

The adaptation policies can be used for specifying reflection or enforcement mechanisms. The notion of reflection means that any unwanted behaviour triggers a corrective reconfiguration through an adaptation policy. The notion of enforcement, exposed in the `AdaptEnfor` algorithm in Fig. 4, means that no reconfiguration that would lead the system to behave in an unwanted way is allowed. This algorithm uses as inputs 1) a generic component-based system *gcbs*—an object used to manage a component-based system regardless of the design/development framework, and, 2) an array, v, containing candidate reconfigurations ordered by priority. Each of the variables $currentConf$, $targetConf$, and $endConf$ represents a configuration while the variable r designates a reconfiguration[11].

This algorithm contains five functions: *a*) `retrieveConf(`s`)` returns the configuration of the generic component-based system s; *b*) `size(`v`)` returns the size

[11] In `AdaptEnfor` Algorithm, $\equiv$ can be implemented by various (pre-)congruence relations—set equality for *Elem* and *Rel* in Def. 1, structural refinement in [12], or other relations compatible with the reconfiguration relation.

```
1  (*AdaptEnfor*)
2  Input
3     gcbs   (*generic component-based system*)
4     v      (*array of candidate reconfigurations ordered by priority*)
5  Variables
6     currentConf,
7     targetConf,
8     endConf: configuration
9     r: reconfiguration
10 Begin
11    currentConf := retrieveConf(gcbs)
12    WHILE (size(v) > 0) DO
13       r := getNextElement(v)
14       remove(v, r)
15       targetConf := applyReconf(currentConf, r)
16       IF (preserveEnforProps(targetConf)) DO
17          applyToSystem(targetConf, gcbs)
18          endConf := retrieveConf(gcbs)
19          IF (endConf ≡ targetConf) DO
20             sendEvent(r, normal)
21             break
22          ELSE
23             applyToSystem(currentConf, gcbs)
24             endConf := retrieveConf(gcbs)
25             IF (endConf ≡ currentConf) DO
26                sendEvent(r, exceptionnal)
27                break
28             ELSE
29                systemExit
30             FI
31          FI
32       FI
33    ENDWHILE
34 End
```

Fig. 4. Algorithm `AdaptEnfor`

of the array v; *c*) `getNextElement(`v`)` returns the next element of the array v; *d*) `applyReconf(`c`,`r`)` returns the resulting configuration when the reconfiguration r is applied to the configuration c; *e*) `preserveEnforProps(`c`)` returns $\top$ if every enforcement property loaded holds on the configuration c, $\bot$ otherwise. Finally, there are also five procedures used within this algorithm: *a*) `remove(`v`,`e`)` removes the element e of the array v; *b*) `applyToSystem(`c`,`s`)` initiates a reconfiguration of the system s to reach a configuration c; *c*) `sendEvent(`r`,`arg`)` sends the event "r **normal**" or "r **exceptional**", where r is a reconfiguration, and arg stands for "normal" or "exceptional"; *d*) `break` exits the current "while" loop; *e*) `systemExit` terminates the current run of the program.

Let us add that the way we enforce properties on adaptation policies supports the *soundness* and *transparency* principles [23]. Given a set of properties to enforce at runtime, the mechanism we use is *a*) *sound* because it prevents (by not entering in the IF statement's body at line 16) the occurrence of reconfigurations that would lead the system to violate, at the next state of execution, the properties to enforce, *b*) *transparent* because it allows (by entering in the IF statement's body at line 16) the occurrence of reconfigurations (if any) that put the system in a state complying with these properties.

The "while" loop starting at line 12 in Fig. 4 ends when the size of the array v becomes equal to 0. Since, on every loop iteration, the size of v is only decremented (line 14), this algorithm always terminates.

Proposition 2. *The* `AdaptEnfor` *algorithm always terminates.*

When the `AdaptEnfor` algorithm terminates with no reconfiguration operation available to be applied to the current configuration, i.e., when the v size becomes equal to 0 in the "while" loop, it means that the set $\{ope \mid (ope \in \bigcup_{A \in AP} R_N) \wedge B_c \wedge G_c \wedge c \xrightarrow{ope}\}(\subseteq readies(c \triangleleft A))$ is empty. In this case, as every adaptation policy for c specifies at least one adaptation rule for a reconfiguration operation, the ready sets of c and $c \triangleleft A$ are different. This way the `AdaptEnfor` algorithm allows answering the adaptation problem with **false**. Moreover,

Theorem 1 (Correctness). *If a configuration c is not reachable in S then, for any AP_S, it is not reachable in $S \triangleleft AP_S$.*

Correctness is clear because when we forget the B and G parts of an adaptation policy A from AP_S restricting a behaviour of $S \triangleleft AP_S$, we get a behaviour of S.

Reflection can be applied in a way similar to the enforcement mechanism presented above. The main difference is that, whereas enforcement prevents the occurrence of specific reconfigurations to avoid unwanted behaviours before they actually happen, reflection allows the detection of such behaviours and triggers corrective actions in the form of reconfigurations performed through adaptation policies. Such actions can range up to the total stop of the system in case of the detection of behaviours that would justify it.

6 Implementation and Case Study

This section describes an implementation developed in Java for the dynamic reconfiguration of component-based systems guided by adaptation policies. A case study shows the result of our experiment on the location component of the CyCab given in Fig. 1.

As shown in Fig. 5, in a nutshell, our implementation uses three controllers: *a*) the *event controller* receives events, stores them, and flushes then after they have been sent to a requester, *b*) the *reflection controller* sends events to the *event controller* when a property of a reflection policy is violated, and *c*) the *adaptation policy controller* manages reconfigurations, as well as, adaptation and enforcement policies as in the `AdaptEnfor` algorithm displayed in Fig. 4.

In addition, an *event handler* is used to receive events from an external source and to send them to the *event controller*. All interactions with the *component-based system* (implemented using Fractal in our case) take place through the *generic component-based system manager*, a set of Java classes developed in such a way that they can be used regardless of the framework used to design the *component-based system* without modifying its code.

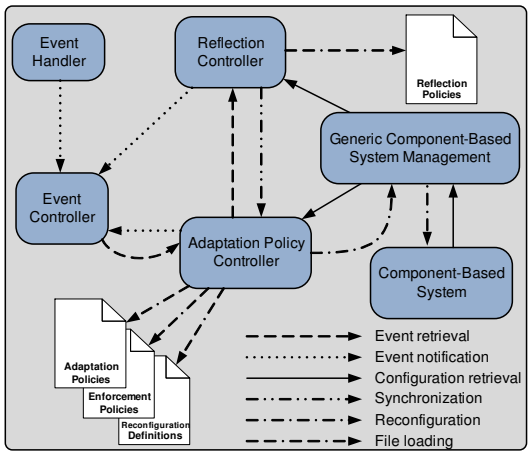

Fig. 5. Implementation architecture

Since we want our implementation to be independent of a particular component-based framework, only a few classes implementing Java interfaces of the *generic component-based system manager* use API specific to the *component-based system* framework. This way, the controllers of Fig. 5 not being Fractal-based, our implementation can manage various component-based frameworks. The synchronization between *adaptation policy* and *reflection* controllers coupled with the way events are managed allows the controllers to operate together under the *perfect synchrony hypothesis* [18].

When running the implementation, each reconfiguration is simulated (cf. line 15 of the `AdaptEnfor` algorithm, Fig. 4), starting with the ones with higher priority. The first one which does not violate any property from an enforcement policy is applied (lines 16 and 17). If the reconfiguration ends normally the event "**`r` normal**" (line 20), where `r` is the name of the above-mentioned reconfiguration, is sent to the *event controller*. If the reconfiguration ends with an error, the previous configuration is rolled back and the event "**`r` exceptional**" (line 26) is sent to the *event controller*. Events sent by the *reflection controller* to the *event controller* are caught by the *adaptation policy controller* that apply corrective actions using appropriate adaptation policies.

Figure 6 provides the results obtained by running our implementation with the location component of the CyCab given in Fig. 1. The top chart illustrates the evolution of the energy level, while the middle (resp. bottom) chart shows the presence (value 1) or the absence (value 0) of the GPS (resp. Wi-Fi) components. Note that the rate of energy consumption is related to the presence or absence of the GPS and Wi-Fi component. When the vehicle enters (resp. exits) a "Wi-Fi area", the event `entry` (resp. `exit`) is sent to the *event controller*, as shown by vertical segments on Fig. 6 at configurations 66 and 134 (resp. 78 and 147).

When the energy level goes below 10, a reflection policy triggers a reconfiguration (`chargeBattery`) that has the effect to update the energy level to 100. Just before configuration 100, a Fractal API has been used to artificially set the energy level to a negative value, this triggers (through a reflection policy) the reconfigurations `stopCycab` and `chargeBattery` that respectively stop the location composite component and update the energy level to 100. The location composite component being stopped, the energy level does not decrease until configuration 116 when it is restarted.

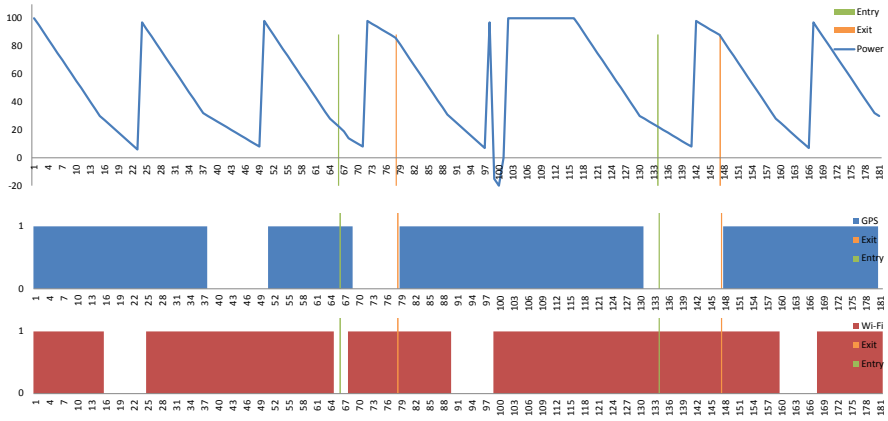

Fig. 6. Experiment with the location component

The `cycabgps` (Fig. 3) (resp. `cycabwifi`) adaptation policy, favours the removal of the GPS (resp. Wi-Fi) component when the energy is low, and favours its addition when the energy level is medium to high. Furthermore, when the CyCab is in a "Wi-Fi area", `cycabgps` (resp. `cycabwifi`) favours the removal of the GPS (resp. addition of the Wi-Fi) component.

At configuration 66, the CyCab enters a "Wi-Fi area" having only the GPS component present. The *reflection controller*, detecting that the vehicle is within a "Wi-Fi area" without the Wi-Fi component, sends, to the *event controller*, the `reflectionNoWifiInWifiArea` event. At the next configuration, as a consequence of the retrieval of this event the *adaptation policy controller* initiates the `addwifi` reconfiguration which adds and starts the Wi-Fi component. Then, at the following configuration, the application of the `cycabgps` adaptation policy (Fig. 3) by the *adaptation policy controller* causes the removal of the GPS component through the `removegps` reconfiguration. At configuration 134, the CyCab enters a "Wi-Fi area" having only the Wi-Fi component present. When the level of energy becomes high (configuration 142), in application of `cycabgps`, the GPS component is not added. As soon as the vehicle exits the "Wi-Fi area" (configuration 79 and 148), since the level of energy is high, the GPS component is added back.

Outside of a "Wi-Fi area", the Wi-Fi (resp. GPS) component is removed at configurations 15, 64, 89, and 159 (resp. 37 and 130), as a result of the application of adaptation policies, because the level of energy becomes low. Still outside of a "Wi-Fi area", when there is only one component present (among the Wi-Fi and GPS components), the other is added when the level of energy is medium to high (configurations 24, 50, 79, 99, 148, and 167).

These experimental results show that extending adaptation policies with temporal patterns provides the specifier with means allowing to better — in comparison with [8,16,9] — comprehend and control the component-based sys-

tem's behaviour. Because the frequency at which adaptation occurs depends on the system under scrutiny, this parameter must be specified as a user-defined para-meter. In a future development it will be possible to specify that adaptation needs to happen in some bounded time.

Of course, it should be possible to come up with a finite encoding of the bounded version of our example to use techniques for finite state systems. Our point, however, is to evaluate temporal and architectural constraints over changeable architectures at runtime. One can imagine new components, not even implemented at the beginning of the run, to be added at execution time. This, indeed, can lead to infinite behaviours.

7 Related Work and Conclusion

7.1 Related work

The analysis of systems whose topology evolves over time is a challenging topic. Tangram4Fractal [8] presents a qualitative approach of adaptation policies, but disallow the use of temporal properties. The work in [9] shows an evolution of Tangram4Fractal that permits adaptation policies based on a qMEDL logic [16] to use external events. Architectural constraints, however, cannot be expressed with qMEDL.

The FTPL logic, expressing temporal and architectural constraints, is introduced in [11]. It is based on Dwyer's work on patterns and scopes [13] and uses specifications inspired by [26]. Nevertheless, this version of FTPL does not support external events and cannot always be evaluated at runtime.

Like Bounded Model Checking [5] (BMC for short), our approach may produce counterexamples when detecting property violation. Moreover, when no violation is detected, both approaches are incomplete for the safety properties. However, for some liveness properties, for example **eventually**, the satisfaction can be established. It is also possible to establish the satisfaction of some safety properties within the appropriate scope [13,26]. Similar to [21] using BMC, we can validate architectural or temporal properties over instantiated reconfigurable systems; this validation is size-bounded and partial.

Evaluation of FTPL properties at runtime is detailed in [10]. This version of FTPL, however, does not support the use of external events. Furthermore, to allow easier runtime evaluation, we use a *progressive* semantics inspired by [3,1]. This semantics, unlike the one in [10], takes fully into account the usage of scopes [13,26]

Our implementation for handling reflection is somehow similar to the steering performed with the MaCS framework in [20]. The coupled PEDL and MEDL scripts act as the event and adaptation controllers in our implementation while the SADL script acts as our reflection controller. No enforcement is provided in MaCS.

In [14], only runtime verification is performed; there is no adaptation mechanism. Nevertheless, by using locations spanning over several components,

the specifications considered for BIP systems allow, similar to our approach, describing global behaviours of the system. Dissimilar to our approach where the code of the component-based system under scrutiny is not modified, RV-BIP slightly modifies components and thus may not allow the component reusability; in this case, the separation of concerns principle would not be respected.

The work in [19] allows runtime monitoring of temporal properties for component interfaces. When components come with an abstract behavioural model, they can be considered as grey boxes rather than black boxes. Our approach, not limited to monitoring interactions of component interfaces with an external application, works in both cases.

7.2 Conclusion

As component-based systems evolve at runtime, and as a behaviour in which the runtime temporal property evaluation becomes false might be not acceptable, this paper has proposed to integrate temporal properties into adaptation policies, and to supervise—at runtime—the reconfiguration execution allowed by the adaptation policies. Inspired by proposals in [4], this paper continues with a four-valued logic allowing to characterize the "potential" properties (un)satisfiability. In addition, the four-valued logic helps in guiding the reconfiguration process, namely in choosing the next reconfiguration operation to be applied. A prototype Java implementation of the algorithm for verifying and enforcing FTPL properties integrated into the adaptation policies has been developed, as a proof of concept.

As a future work, we plan to investigate a decentralized method to evaluate adaptation policies and temporal formulae by progression, as in [3]. On the implementation side, a future direction is to handle component-based systems using the FraSCAti [25] framework.

References

1. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Annals of Mathematics and Artificial Intelligence 22(1-2), 5–27 (1998)
2. Baille, G., Garnier, P., Mathieu, H., Pissard-Gibollet, R.: The INRIA Rhône-Alpes CyCab. Tech. Rep. RT-0229, INRIA (apr 1999)
3. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. In: FM 2012: Formal Methods. LNCS, vol. 7436, pp. 85–100. Springer (2012)
4. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. Journal of Logic and Computation 20(3), 651–674 (2010)
5. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in computers 58, 117–148 (2003)
6. Bloom, B., Istrail, S., Meyer, A.R.: Bisimulation can't be traced. In: Ferrante, J., Mager, P. (eds.) POPL. pp. 229–239. ACM Press (1988)
7. Chang, E., Manna, Z., Pnueli, A.: Characterization of temporal property classes. In: Kuich, W. (ed.) Automata, Languages and Programming, LNCS, vol. 623, pp. 474–486. Springer Berlin Heidelberg (1992)

8. Chauvel, F., Barais, O., Plouzeau, N., Borne, I., Jézéquel, J.: Composition et expression qualitative de politiques d'adaptation pour les composants Fractal. In: Actes des Journées nationales du GDR GPL 2009 (Jan 2009)
9. Dormoy, J., Kouchnarenko, O.: Event-based Adaptation Policies for Fractal Components. In: Computer Systems and Applications, 2010. AICCSA 2010. IEEE/ACS International Conference on. pp. 1–8. IEEE (May 2010)
10. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Runtime verification of temporal patterns for dynamic reconfigurations of components. In: Arbab, F., Ölveczky, P. (eds.) FACS, LNCS, vol. 7253, pp. 115–132. Springer Berlin Heidelberg (2012)
11. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: Barbosa, L., Lumpe, M. (eds.) FACS, LNCS, vol. 6921, pp. 200–217. Springer Berlin Heidelberg (2012)
12. Dormoy, J., Kouchnarenko, O., Lanoix, A.: When structural refinement of components keeps temporal properties over reconfigurations. In: FM 2012: Formal Methods, pp. 171–186. Springer (2012)
13. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE. pp. 411–420 (1999)
14. Falcone, Y., Jaber, M., Nguyen, T.H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems. In: Software Engineering and Formal Methods, pp. 204–220. Springer (2011)
15. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of safety-progress properties. In: Bensalem, S., Peled, D. (eds.) Runtime Verification, LNCS, vol. 5779, pp. 40–59. Springer Berlin Heidelberg (2009)
16. Gonnord, L., Babau, J.P.: Quantity of resource properties expression and runtime assurance for embedded systems. In: Computer Systems and Applications, 2009. AICCSA 2009. IEEE/ACS International Conference on. pp. 428–435. IEEE (2009)
17. Hamilton, A.G.: Logic for mathematicians. Cambridge University Press, Cambridge (1978)
18. Jantsch, A.: Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation. Morgan Kaufmann (2004)
19. Kähkönen, K., Lampinen, J., Heljanko, K., Niemelä, I.: The lime interface specification language and runtime monitoring tool. In: Runtime Verification. pp. 93–100. Springer (2009)
20. Kim, M., Lee, I., Shin, J., Sokolsky, O., et al.: Monitoring, checking, and steering of real-time systems. ENTCS 70(4), 95–111 (2002)
21. Lanoix, A., Dormoy, J., Kouchnarenko, O.: Combining proof and model-checking to validate reconfigurable architectures. ENTCS 279(2), 43–57 (2011)
22. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM TISSEC 12, 19:1–19:41 (January 2009)
23. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. Int. J. of Information Security 4(1-2), 2–16 (2005)
24. Manna, Z., Pnueli, A.: A hierarchy of temporal properties (invited paper, 1989). In: Proc. of the 9th ACM Symp. on Principles of distributed computing. pp. 377–410. ACM (1990)
25. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A component-based middleware platform for reconfigurable service-oriented architectures. Software: Practice and Experience 42(5), 559–583 (2012)
26. Trentelman, K., Huisman, M.: Extending jml specifications with temporal logic. In: AMAST 2002. LNCS, vol. 2422, pp. 334–348 (2002)